## **EXPRESSIS**

## Express

### **Overview**

What is <u>Express</u>? Well, <u>Express is just a library/framework built for Node.js that allows us to more easily handle requests and build a robust server.</u>

There are tons of ways to build apps with Express! Unlike other MVC frameworks (like Codeigniter and Rails), Express is quite flexible in the way it can be configured. This provides tons of customization for experienced developers. However, for someone more on the beginner side, Express can seem like an endless maze of Javascript. In this section, we will show you the easiest way to configure Express right out of the box so we can start building awesome apps. Keep in mind that there are plenty of other ways to configure Express but the fundamentals presented here are more than enough to grasp the core concepts!

We are going to build Express applications from scratch, so you will need to have a working knowledge of the following before starting this chapter:

- Client-Server Model
- The Traditional HTTP request/response cycle
- What is Node.js?
  - o What does "require" do?
  - What is module.exports?
  - o What are Node modules (middleware)?

## Express

## What is Express?

**Express.js** is a framework built in Javascript using Node.js as the server component. It is also the **E** in the **MEAN** stack, so it must be important! Unlike most other MVC frameworks (like Codeigniter and Rails), **Express isn't exclusively MVC**. It's actually more so just a set of tools that allows us to create a more robust Node Server. There are a few different ways to build an MVC framework with Express, and **we will eventually show you how we do it**. However, we believe the best way to learn **Express** is to start with the basics before working our way up.

### Your first Express project:

Create a folder called "Hello\_Express" and give it a file called "server.js". This file will be our Node server file.

Since Express is a node module, run npm install express while in the Hello\_Express directory.

Note: If you are on a Mac you may need to run this command as sudo.

All that this did was go out and install the express module (as well as all of the modules that express is dependent on) in your project folder. Your project folder should now contain a big new scary "node\_modules" folder in addition to your server.js file! Feel free to poke around and see what is inside, but do not delete or change anything. Try to trace where require("express") from our server.js code goes and what it comes back with.

### Hello\_Express/server.js

```
// Load the express module and store it in the variable express (Where do you think this comes from?)
var express = require("express");
console.log("Let's find out what express is", express);
// invoke express and store the result in the variable app
var app = express();
console.log("Let's find out what app is", app);
// use app's get method and pass it the base route '/' and a callback
app.get('/', function(request, response) {
    // just for fun, take a look at the request and response objects
    console.log("The request object", request);
    console.log("The response object", response);
    // use the response object's .send() method to respond with an h1
    response.send("<h1>Hello Express</h1>");
})
// tell the express app to listen on port 8000, always put this at the end of your server.js f
ile
app.listen(8000, function() {
    console.log("listening on port 8000");
})
```

Run the server is file using node (but use nodemon, it will make your life easier) and see the magic happen!

nodemon server.js

## **Views**

### What does the client see?

HTML/CSS/JS! Remember that the client only sees HTML/CSS/JS? Generally, there are two ways to serve HTML/CSS/JS - through Static Content or Templates.

**Static Content** -- Serving a static HTML/CSS/IS file from the backend in response to a request.

**Templates** -- Using a view/templating engine to generate HTML (PHP, embedded ruby, embedded JavaScript)

### Serving Static Content

Remember that static content module you made in the Node.js chapter? Well, Express has one of its own that is really
easy to use! Just like we wrote the instructions on how to handle the "/" route in our server.js file we are going to add a
line that points our express server to a static files folder.

First create a new folder inside your project and call it "static" and place an HTML file called "main.html" in it. Add some content to the file so you can see it render!

Next open your server.js file and let's tell our server to use a static file folder to handle requests for static content!

```
// this is the line that tells our server to use the "/static" folder for static content
app.use(express.static(__dirname + "/static"));
// two underscores before dirname
// try printing out __dirname using console.log to see what it is and why we use it
```

Now after restarting the server, you should see the HTML page you created when you navigate to

"localhost:8000/main.html". **It is important to note that now all of your static content must go in the static folde**including styles and static javascript files. Try adding a CSS file to show that you really understand how the static

content works!

### Using Templates (EJS)

EJS is the templating engine we are going to use. EJS stands for Embedded Javascript. Just like express, ejs is a node module that we will need to install for our project.

First let's install the eis module in our project.

### npm install ejs

Next let's tell express that we are going to use eight

```
// This sets the location where express will look for the ejs views
app.set('views', __dirname + '/views');
// Now lets set the view engine itself so that express knows that we are using ejs as opposed
to another templating engine like jade
app.set('view engine', 'ejs');
```

Let's say we wanted to add a route to our app that displays a list of users. We aren't going to get our list of users from the database, we're just going to hard code our data for now. Let's add a new route to our server.js file that will render an ejs view and pass it some user data.

**Notice we are passing a JavaScript object to the response.render()** method. That way, when we pass a piece of data to a view, every key-value pair within the larger piece of data becomes its own variable.

Let's create a new folder called "views" in our project directory. This folder is where we are going to put all of our view

Next let's make a view called **users.ejs** and put in it the following:

The <% %> tags are the delimiter for the embedded JavaScript. Using these tags allows us to run JavaScript code that can be embedded into the HTML document we are making. Notice the <% %> tags allow us to enter JavaScript code, and the <%= %> tags actually print the JavaScript code to the document. This is a key difference. You'll use the tags with the equal sign (=) to actually print values, whereas you'll use the tags without the equals sign to invoke loops or use logic (anything that involves JavaScript but doesn't output code). Embedded JavaScript should be pretty quick to get up and running out of the box. Play around with it and move on when you're ready!!

## **Express Cars and Cats**

You now have the tools necessary to repeat the Cars and Cats assignment from Node, but with Express!

Take the time to appreciate what a difference a framework makes.

For this assignment, you will need a static directory. You will not need routes, ejs, nor a views directory.

we're requesting static content, and because of our Express static middleware, our server knows to find static files in the static directory.

localhost:8000/cars.html - A simple HTML page that shows some cool pictures of different cars. These car pictures should be stored in your static directory. DON'T just link to pictures of cars stored somewhere else! Even better, put them in a directory called 'images' inside of your static directory.

localhost:8000/cats.html - A simple HTML page with some cool pictures of cats. Again, make sure these pictures are stored on your server.

localhost: 8000/form.html - A simple form where the user can add new car information. For this page, there is no need to have the form do anything. Simply display the form there.

Also, add a basic html file in your static directory called index.html. What happens when you navigate to the root route localhost:8000?

## EJS Cars and Cats

Repeat the previous assignment, but this time use EJS and include a views directory.

By telling Express where to find your views directory and that you are using EJS, you will be able to write the following routes in your server:

Have '/cars' show your pictures of cars.

Have '/cats' show your pictures of cats.

Have 'cars/new' show a form to create a new car. The form does not have to do anything yet.

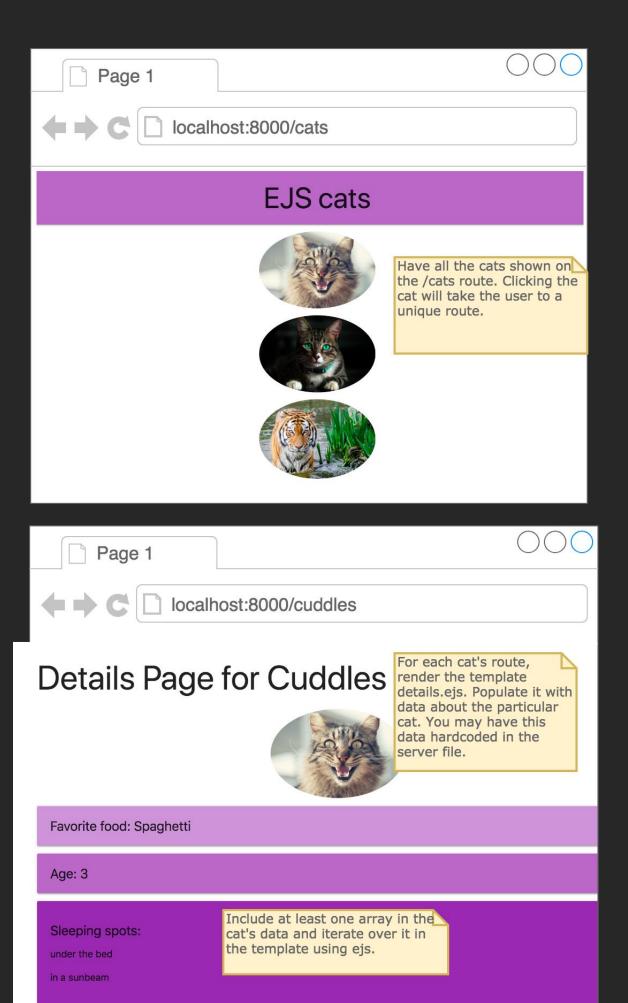
Keep your index.html file in the static directory. It should render even when your server does not explicitly handle the '/' route

## **EJS Cat Data**

In this assignment, we are going to focus on displaying data about our particular cats

As before, when we navigate to localhost: 8000/c ats, we should see pictures of all our cats. This time, when we click the pictures, they will navigate us to a unique route in the server. **This route may be hardcoded for now.** 

For each of the cat's routes, serve the same template, details.ejs. However, each route should send different data to the template so that the page is customized for each cat. Again, **this data may be hardcoded in the server** since we do not have a database yet. Include at least one array in the cat's data so that you may practice iterating over an array in the template.



## HTTP Methods

### Forms, Data Transferring, and Routing Rules

The modern internet is user-driven; all the actual content of a website is generated by the users of a website. How does the user manage to get his or her data to the page?? One word: **forms.** The HTML form is a way in which users are able to pass data to the back end of a website where data can be processed and stored. Designing forms and processing the information correctly is a huge part of what it takes to become a back-end developer.

HTTP Methods: GET and PC

If you are not already familiar with *HTTP request methods*, here's a brief run-down on the two most common: **GET** and **POST**.

- GET is used for passing insensitive information
- POST is used for passing sensitive information.

When you create a form tag in HTML, you always need to specify the *method* used to send the request. The method attribute specifies the type of HTTP request by how you, as the developer, want to have your form information transferred. If your form data contains password data, credit card numbers, SSNs, etc, please use **POST**! If the information is like a library search or something insensitive, it's OK to use **GET** as your method.

### **GET**

GET requests are sent as part of the URL. Ever noticed that when you do a Google search, your search term appears in the resulting URL? That's because it's a GET! That wouldn't be very secure for passwords! Imagine if every time you logged into Gmail, it displayed your password at the top of the screen, right in your URL for the world to see! Worse still, **GET requests can be cached, and will even remain stored in your browser's history!** So why do we use GET at all?? Well one neat thing, GET requests can be bookmarked! This lets you bookmark routes on say Google Earth or image results on a search engine.

### POST

POST requests send data behind the scenes, in the HTTP message body. They're never cached, they don't linger in your browser history, they can't be bookmarked, and there are **no restrictions on how much data you can send.** The vast majority of HTTP request methods you'll be dealing with will be POSTs. Forms that control user registration, user authentication, user authorization, database inserts, updates, and deletes, all of these will be sent by the POST method.

## POST, GET, & SESSION

### Routing

In Express we will assign all of our routing rules using the syntax:

```
app.HTTP_VERB('URL', function (req, res){}); // HTTP_VERB is either 'get' or 'post' etc...

So let's say we were working in a RESTful environment and we wanted to make the route for creating a new user. The setup would look like this:
```

```
// root route
app.get('/', function (req, res) {
   res.render('index', {title: "my Express project"});
});
// route to process new user form data:
app.post('/users', function (req, res) {
   //code to add user to db goes here!
})
```

Easy! Remember that all of these routes must go above where we tell the server to listen.

### Redirecting

After we do anything with the handler of the **POST request**, provided there are no errors, **we need to redirect.** Redirecting is easy! Let's revisit the routes we made above and have the code redirect the user back to

```
// root route
app.get('/', function (req, res){
   res.render('index', {title: "my Express project"});
});
// route to process new user form data:
app.post('/users', function (req, res){
   // code to add user to db goes here!
   // redirect the user back to the root route.
   // All we do is specify the URL we want to go to:
   res.redirect('/');
})
```

### POST data

In order to be able to access POST data, we need to be able to pull it out of the request object. Unfortunately in an effort to be as lightweight and flexible as possible, Express doesn't have a good way to handle post-data inherently. So how do you think we are going to get our data from the request object? A node module! This one is called body-parser.

After installing body-parser (you should be very familiar with npm install by now), require it and tell the express server to use it like so:

```
// require body-parser
var bodyParser = require('body-parser');
// use it!
```

```
app.use(bodyParser.urlencoded({extended: true}));
```

Let's say in an **index.ejs** view file we had a form that looked like this:

```
<form action='/users' method='post'>
   Name: <input type='text' name='name'>
   Email: <input type='text' name='email'>
   <input type='submit' value='create user'>
</form>
```

We're in the **MEAN** stack, right? JavaScript runs the universe now. What do we expect our form data to be? You guessed

```
// route to process new user form data:
app.post('/users', function (req, res) {
    console.log("POST DATA \n\n", req.body)
    //code to add user to db goes here!
    // redirect the user back to the root route.
    res.redirect('/')
}):
```

Now we will go to the **terminal window our server is running on** and check it out...

There you have it! req.body is a JSON object that contains the data from our form. Not so bad.

### Data from URL (GET data)

Let's say we wanted to show the information of a specific user. The **RESTful route** for this would be:

### users/:id //where :id is the id of a particular user. HTTP method is GET

Setting this up in Express is easy; in our **server.js** file we would just **add the route**:

```
app.get("/users/:id", function (req, res) {
   console.log("The user id requested is:", req.params.id);
   // just to illustrate that req.params is usable here:
   res.send("You requested the user with id: " + req.params.id);
   // code to get user from db goes here, etc...
});
```

Note: This illustrates **accessing data from the URL.** If you want to test this out, **add a button on the view that sends a request to "/users/1"** (trying to access the page with information about the user with id = 1). Note that a post request to "/users" and a get request to "/users" are two completely different routes because a route is made up of the **verb + the URL.** 

Any data you wish to pass via the URL must be indicated by a ':'. It will then be available in the req.params object.

### Session data

Session data should be used as little as possible (depending on the situation); using the proper request and response cycle, you will find that there is rarely a need for session data. Using session where unnecessary incurs needless overhead and is heavily discouraged. We teach it briefly here to give you some exposure to it, but it is heavily

recommended against at this stage. <u>If you do need to use it, you have to install and require the express-session module first.</u> Common use cases include 'logging in' a user, and storing their 'user\_id' into session to be able to retrieve them from different routes

After installing go to your server.js file and require it like so:

```
// new code:
var session = require('express-session');
// original code:
var app = express();
// more new code:
app.use(session({
   secret: 'keyboardkitteh',
   resave: false,
   saveUninitialized: true,
   cookie: { maxAge: 60000 }
}))
```

Now, within any of your routes, there will be an object called **req.session**. It is an object and you can assign properties to it like normal, in this example, we are storing into req.session.name the value of the post data req.body.name:

```
app.post('/users', function (req, res){
    // set the name property of session.
    req.session.name = req.body.name;
    console.log(req.session.name);
    //code to add user to db goes here!
    // redirect the user back to the root route
    res.redirect('/');
});
```

And now **reg. session name** will be available to any other route afterward (until our session expires, more on this later)

## package.json, bower.json

Package managers, such as NPM and Bower, keep track of what modules you are using through JSON files. In general, these files contain information about the project overall, as well as which modules have been downloaded for the project, specifically. If you choose to move a project from one machine to another e.g. into GitHub (or submitting to the site), these .json files minimize the need for copying all of the dependent modules that we download from the package management sites, keeping the overall size of the project that we pass much smaller.

We have dealt with three packages that we include in a lot of our files so far that aren't pre-installed with node: express, ejs, and body-parser. Let's use NPM's package.json file!

### Create an empty folder called TestProject

- navigate to that folder using your terminal/command-prompt/bash.
- in terminal/command-prompt/bash type npm init -y

o this command basically says: we are using this folder for an npm based project, (npm init), the (-y) says fill the package.json with the base information.

- Now let's install ejs and express, from command-line in the same folder:
  - o npm install express --save
  - npm install ejs --save
  - npm install body-parser --save
- Now look at how that package.json has changed

```
"name": "TestProject",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
},
"keywords": [],
"author": "",
"license": "ISC",
"dependencies": {
    "body-parser": "^1.15.0",
    "ejs": "^2.4.1",
    "express": "^4.13.4"
}
```

You should also have a new node modules folder with those packages inside.

### Want to use the same dependencies for another project?

- You have some options, but our favorite way is: Create the new project, npm init -y in the new project, then copy the dependencies to that new package.json.
- Then type: npm install. Install, when not passed any information, will look for a package.json file and then install all the listed dependencies in the package.json.

Bower works almost identically: however, you have to manually type in the initialization conditions, rather than using -y. Just hitting the enter key at each question will initialize the bower.json with the default conditions.

This is just a taste of what npm can do from the command line: for more see <a href="https://docs.npmjs.com/cli/npm">https://docs.npmjs.com/cli/npm</a>

To explore bower: http://bower.io/

## **Dependencies Intro**

Learning the MEAN stack requires learning the individual parts piece by piece, and then combining them into one big whole. It's because of this that you will need to utilize different Node.js modules contingent on what part of the MEAN stack you're working on. These are your **dependencies**. As an example, when you get to the Angular chapter, your node\_modules folder and your package.json will shrink down to just requiring express!

If you're ever unsure of what dependencies are required for the chapter you're currently working on, check the dependencies tab for a comprehensive list. The most important thing to understand is that **you won't just be adding** to your dependencies, **but instead it will be an ever expanding and contracting list!** 

## Dependencies for Express & Socket.io

Node.js

This will be a constant across the vast majority of the MEAN stack. You will almost always be running a Node server or app of some kind. This will act as your server and back-end Javascript interpreter.

Express

Technically a Node module, but let's talk about it here too since it gets its own letter in the acronym! Express is a wonderful set of tools that help us write rules for incoming HTTP requests. Express is arguably the most useful Node module in your collection, and only rarely will we not use it.

- node\_modules:
  - o EJS

Embedded Javascript is the templating engine we will use to render our views. When using EJS, your node server will take the .ejs file you wrote, parse through it, resolve all of the Javascript in the file, and then send a 'rendered' HTML page to the client. **This is extra computation that the server will be required to do**, and later we'll replace EJS with a full front end framework like Angular.

o Express

Like we said before, express is super useful for managing incoming requests!

### o **body-parse**r

Guess what body-parser is used for? **Parsing the body!** We'll use this piece of middleware to parse information out of HTTP requests made to our server. Body-parser is incredibly flexible, and will be used to not only pull POST data out of requests, but can also snag data encoded into URLs via GET requests, and later you'll use body-parser to pull raw JSON.

### o socket.io

This node module will enable you to use web sockets within your application, we'll get into it more later You will use socket.io for just this chapter!

## package.json

You can use this package.json as a guide, but we strongly recommend using npm to bring in your Node modules for every project you make. Typing *npm install express --save* will make sure you get the newest version of express!

```
"dependencies": {
    "body-parser": "^1.13.3",
    "ejs": "^2.3.3",
    "express": "^4.13.3",
    "socket.io": "^1.3.6"
}
```

## Setting up a full server!

Create a new project folder.

Copy this into a package. Json file inside that folder

```
"name": "TestProject",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
},
"keywords": [],
"author": "",
"license": "ISC",
"dependencies": {
    "body-parser": "^1.15.0",
    "ejs": "^2.4.1",
    "express": "^4.13.4"
```

```
]
```

Copy this into a new server.js file inside that folder:

```
// require express
var express = require("express");
// path module -- try to figure out where and why we use this
var path = require("path");
// create the express app
var app = express();
var bodyParser = require('body-parser');
// use it!
app.use(bodyParser.urlencoded({ extended: true }));
// static content
app.use(express.static(path.join(__dirname, "./static")));
// setting up ejs and our views folder
app.set('views', path.join(__dirname, './views'));
app.set('view engine', 'ejs');
// root route to render the index.ejs view
app.get('/', function(req, res) {
   res.render("index");
})
// post route for adding a user
app.post('/users', function(req, res) {
   console.log("POST DATA", req.body);
// This is where we would add the user to the database
// Then redirect to the root route
   res.redirect('/');
})
// tell the express app to listen on port 8000
app.listen(8000, function() {
   console.log("listening on port 8000");
});
```

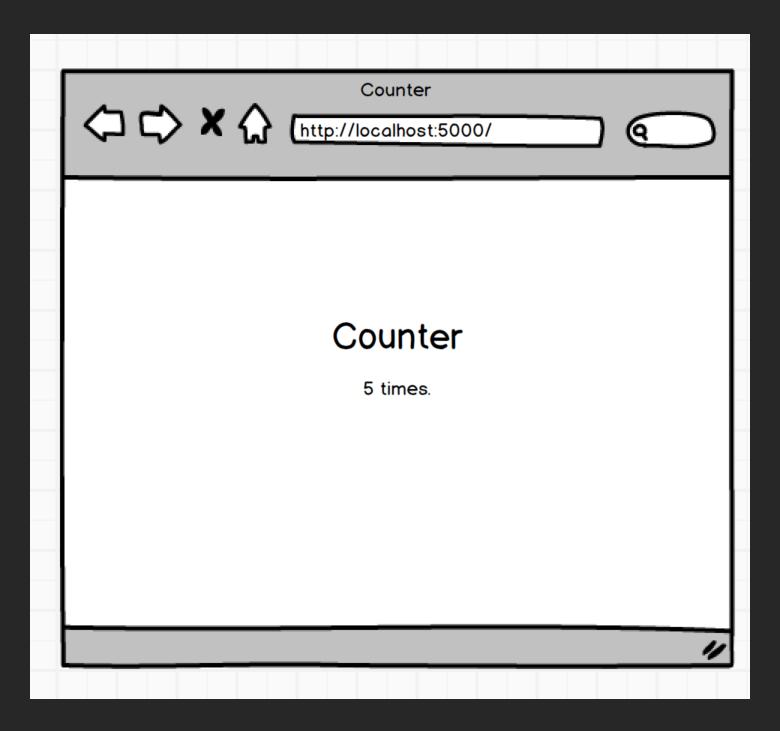
navigate to this folder with your command line, and then type npm install.

create an index.ejs file (and place it in the appropriate place based on the server) to load a page when you go to localhost:8000.

### **Assignment: Counter**

Build an Express application that counts the number of times the root route ('/') has been viewed by the user.

This assignment is to test your understanding of session.



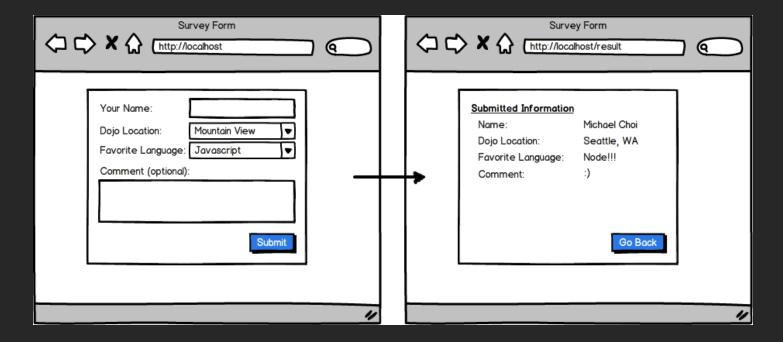
### Ninia Level 1

Add a +2 button underneath the counter that reloads the page and increments counter by 2. Add another route to handle this functionality.

### Ninia Level 2

Add a reset button that resets the counter back to 1. Add another route to handle this functionality.

### **Assignment: Survey Form**



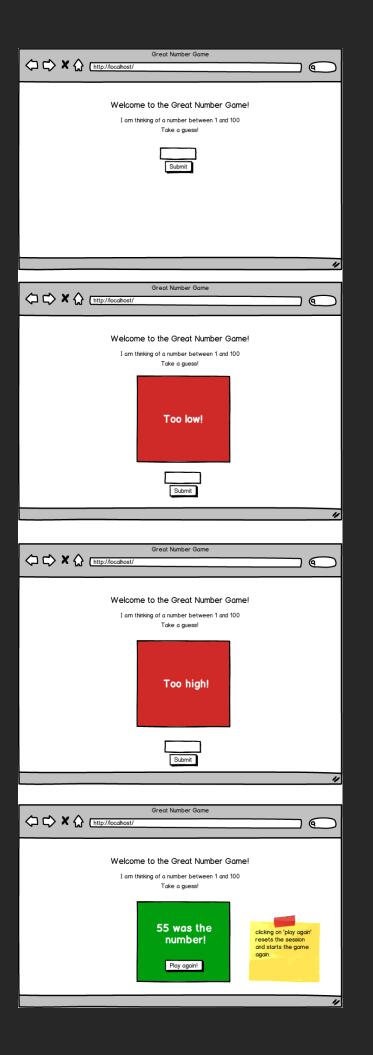
Before you start coding this, first outline or write down the steps of accomplishing this.

### For example

- 1. Have the server render views/index.ejs that has the form for the user to fill out
- 2. The user fills out the form and submits
- 3. The submitted form gets sent to /result
- 4. The server recognizes when someone posts things to /result, grabs information from the POST, and sends the POST data back as it renders views/results.ejs

### Assignment: Great Number Game

Create a site that when a user loads it creates a random number between 1-100 and stores the number in **session**. Allow the user to guess at the number and tell them when they are too high or too low. If they guess the correct number tell them and offer to play again.



Note: There are many different ways to do this assignment. When you finish the basic functionality, find a peer and compare your code!

## Adding AJAX and working with CORS

## **Objectives:**

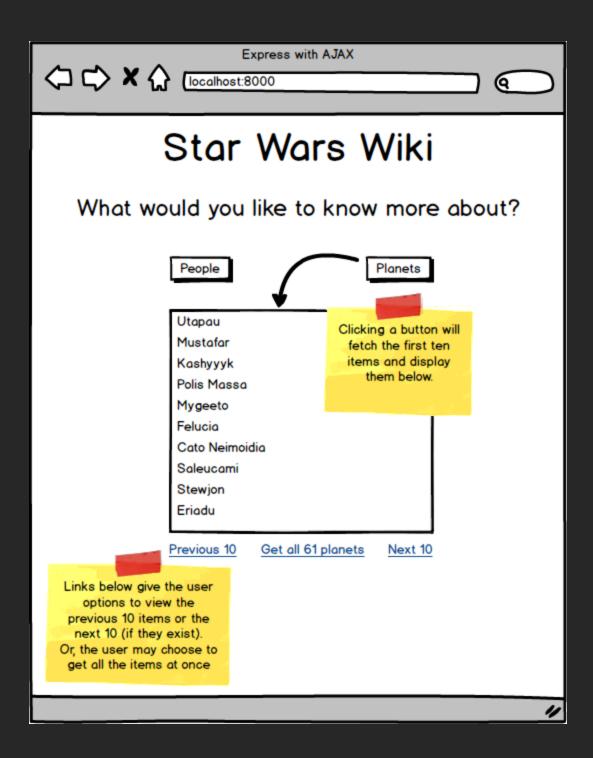
- 1. Review AJAX
- 2. Include AJAX with your Express projects
- 3. Understand Access Control Origin

Let's talk about AJAX again! With the assignments we have done so far with Express, we have been focusing on server-side rendering. With the help of embedded JavaScript, we are able to have our server construct HTML based on its current data. When the HTML is ready, the server sends it to the client. This is an essential piece of how the Internet works, but you may have noticed that this may lead to an unsatisfactory user experience.

When we have to render the entire HTML document that will be delivered to the client, this causes excessive refreshing of the page. Sometimes we do not want to change the entire document. For example, imagine adding a comment to a Youtube video. Should it really be necessary to rebuild the entire page's HTML just to show our comment has been added? If we were watching the video as we were posting, the refresh would cause our video to restart at the beginning. How terribly annoying!

# AJAX allows us to manipulate the DOM so only small pieces of the HTML have to change

We know how to solve this excessive refresh problem. It's by using AJAX! We've done this before. Let's go through the process that we would need for the wireframe below, which gets its data from the **Star Wars API**:



Previously, we used jQuery to make requests to an API. Therefore, we may decide to have our server send index.ejs to the client, and we could have index.ejs import jQuery and make requests to the API when the button is clicked. The code below shows one way this could be done:

### index eig

However, what happens when we try this? Sadly, we'll see this message in our console:

```
    Sailed to load <a href="http://swapi.co/api/people">http://swapi.co/api/people</a> to 'https://swapi.co/a swapi.html:1 <a href="pi/people">pi/people</a> has been blocked by CORS policy: No 'Access−Control−Allow−Origin' header is present on the requested resource. Origin 'null' is therefore not allowed access.
```

## Cross-Origin Resource Sharing (CORS) and Access Control Origin

The access-control-origin error is caused by a configuration setting on certain API servers, where the API provider has configured the server to **control** incoming requests. It allows only those from certain origin types to **access** the API. This means that our client-side JavaScript code cannot directly contact the API server with requests.

To bypass this, we have to make an AJAX request from our client-side page *to our own server*. Notice that in the code snippet below, the button click is triggering a request to the route '/people', which is a route we make on our own server. Let's have our **server** make the request to the API and pass back the response.

index.eis

To make requests from our server to different servers, we can use the **Axios** package. It is not the only option available to us, but this one is nice because it uses **promises**.

To use a promise, we'll need two **callbacks**: one if the request is successful, and another if it fails. We place these callbacks in the .then() and .catch() methods respectively, which are chained after the request.

Visit the optional <u>JavaScript Advanced chapter</u> to learn more about promises if you haven't done so already.

To get started, use npm to install Axios, and then require it into your server

### **Arrow functions**

In the code snippet above, we are using ES6 **arrow functions**. They are a more concise way to write anonymous functions! Remember that anonymous functions do not have names, but they may be stored in a variable. Depending on the code within them, parentheses, curly brackets, and the return statement are not always necessary. However, it does not hurt if you choose to include them.

```
// es5 style
var anonES5 = function(parameter) {
    return parameter + 5;
}
// arrow functions
const anonES6 = parameter => parameter + 5;
// curly brackets are not required if there is only one expression
// parentheses are not required if there is only one parameter
// the return is implicit with just one line
const twoParams = (parameter1, parameter2) => {
    parameter1 += 5;
    return parameter1 + parameter2;
}
// with more parameters, parentheses are required
// with more lines of code, curly brackets are require
```

We will be using arrow functions extensively when we work with Angular because they do not bind their own this, so

the meaning of this is more intuitive. Arrow functions use what is called the lexical

this, which means this will always refer to the this of the code that contains the arrow function.

## Star Wars API

## **Objectives**

- 1. Gain familiarity with CORS and access control origin
- 2. Make requests to an API from the server
- 3. Make recursive API calls

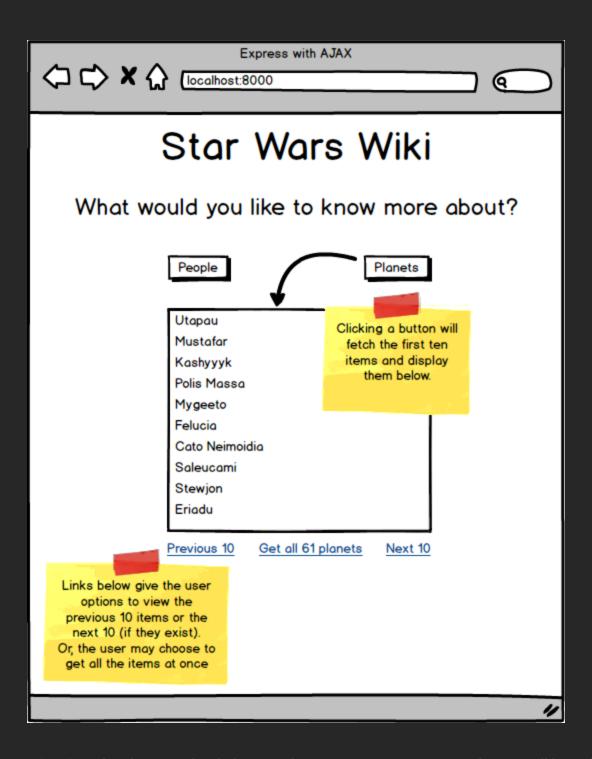
Use the <u>Star Wars API</u> to gather information about the people and planets of Star Wars based on what the user requests.

Upon first visiting the web page, the user should see two buttons - one to receive information about people, one to receive information about planets.

Remember that we should not refresh the entire page. Instead of rendering a new page, have your server respond with JSON by using res.json(). The client should use the JSON data to manipulate the DOM, which means only a portion of the page needs to be updated.

You will notice that the API gives data in chunks. For example, making a request to <a href="https://swapi.co/api/people">https://swapi.co/api/people</a> will respond with an object with an array that contains only ten people. The <a href="count">count</a> attribute, however, tells us that there are 87 people we could access. We need to follow the url provided in the <a href="next">next</a> attribute to get the next ten.

This assignment asks you to provide links that allow the user to fetch the next ten or previous ten results. Additionally, provide a link that fetches all the people or all the planets! This will require recursion. Notice that this will take a while to load. The intention is to give you experience with recursion and response times so that you may make wise choices about user experience in the future.



**BONUS:** Rather than providing links to get the next or previous ten items, place a scroll bar to look at the list of fetchec items. When the scroll bar hits the bottom of the list, make an API call to fetch the next ten and append them to the list.